



The meaning of life and scope

Part 2

Last month, we discussed the life and scope of an object. Life refers to when an object comes into being, a run-time concept. Scope represents the opportunity for access, a compile-time concept. This month, we'll look at some issues and peculiarities in life and scope, examine the external and static storage, initializers, and the uses of `extern` and `static` keywords.

Life and storage classes

Any object—variable or function—has a life span based on its storage class. There are only two storage classes for determining life—automatic or static. An automatic variable springs into life during the program's execution. These variables are dynamic and mortal—like the karma chameleon, they come and go. A static object is immortal in the sense that the object's life span is identical to that of the program. The object's life span starts when the program starts and ends when the program ends.

A data type with an `auto` or `register` storage class is mortal—the data is dynamically created and terminated. Any object with the `static` storage class is immortal, lasting as long as the program. However, all objects defined outside a function (external objects) are also immortal. This description includes any data type, simple or aggregate, defined outside a function and all function definitions.

This discussion also points out a similarity between C concepts and C operators; i.e., both are overworked. Just as the equal sign has two meanings in C (assignment and relational equality), `static` and `extern` have two meanings. There is a static storage class and a static storage class keyword. There is an `extern`, or external storage class, and an `extern` keyword. We'll later show the differences between the concept and contextual use of these words.

When does life begin?

For static variables (globals, functions, and intrafunction variables declared with the `static` keyword), life begins when the program begins. These static variables' lives are the same as the program's life. From that standpoint, these variables are immortal.

Automatic variables live and die. When the birth and funeral take place is a subject for the implementer.

Figure 1 is a code fragment that can show a property of compilers. `MAXARR` is defined as 1/4 of the possible data segment (which is not true on all machines, but is good for an example). Three arrays are defined in the function. The question is, How much automatic storage space will the function use?

The answer is either 16K or 48K, depending on when the compiler births and kills automatic variables. Both answers are correct and possible.

K&R loosely defines that the life of an automatic variable begins each time the function or block is entered. K&R *hint* that automatic variables inside a block are created when the block is entered. However, this is usually not true.

Most compilers dynamically allocate the space for a function's automatic variables when the function is entered. The rules for lexical scope "hide" the variables in previously exited scope levels so that implementation remains pure (so you don't access a "dead" variable). For these compilers, the correct answer is 48K. And on many machines, this function would cause the compiler to gag.

In this issue

• The meaning of life and scope - Part 2	1
• Letters to the editor	2
• Corrections and clarifications	3
• ANSI committee releases tentative form of C language draft	9
• Quickie	9
• <code>fd</code> versus <code>fp</code>	9
• C quiz	16
• D language	16
• Quickie answer	16

(Continued on page 4)

Letters to the editor

Q. What is C++? I have heard some references that it's a superset of C, but I don't have any other information.

Jerry Bailey
Boston, MA

A. C++ is a superset of the C language. Designed at Bell Laboratories, C++ is now running at about 120 UNIX installations. It was recently released as a commercial product by AT&T.

Some major features of C++ are Simula-like classes (Simula is a programming language), data hiding capabilities, implicit type conversions for user-defined objects and data types, and mechanisms for creating user-definable operators.

When compared to C, the most foreign concepts of C++ are Simula-like classing and "creating" data types. Groups of variables (from simple to aggregate to user-defined data types) and functions can be declared as a class. Only functions declared in an object class can access those variables. New data types, such as complex numbers, can be created and manipulated.

Don't look for C++ everywhere. The current compiler is too large to work on any member of the PDP-11 family that features a 128K-work space. C++ needs about three times that work space amount, meaning that this C-superset cannot currently run on most micro- and minicomputers.

/c: The Journal for C Users is published monthly by

Que Publishing, Inc.
7999 Knue Road
Indianapolis, Indiana 46250

Address all editorial correspondence to the Editor, /c, at the address above. All requests for special permission should be addressed to the Publisher, /c, at the same address.

Domestic subscriptions: 12 issues, \$60.00; outside U.S.A., \$80.00. Single copy price, \$5.00; outside U.S.A., \$7.00. All funds must be in United States currency. Application to mail at second-class postage rates is pending at Indianapolis, Indiana. POSTMASTER: Send address changes to: /c, 7999 Knue Road, Indianapolis, Indiana 46250. READERS: Send subscriptions, change of address, fulfillment questions, or bulk orders to

/c Subscriptions
P.O. Box 50507
Indianapolis, Indiana 46250

Copyright© 1985, Que Corporation. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever, except in the case of brief quotations embodied in critical articles and reviews, without the prior written consent of Que Publishing, Inc.

Publisher: Patty Stonesifer
Editor-in-chief: Chris DeVoney
Managing Editor: T.L. Solomon
Assistant Editor: PJ Schemenaur
Circulation Coordinator: Sharon Sears
Production Designer: Dan Armstrong
Cover Design: Don Tiemeier of Listenberger Design Associates

/c and Que are trademarks of Que™ Corporation. IBM is a registered trademark of International Business Machines Corporation. MS-DOS is a trademark of Microsoft™, Inc. UNIX and AT & T are registered trademarks of American Telephone and Telegraph Corporation.

If you're interested in more information, the C++ manual is available from AT&T. Also, an excellent article on C++ appeared in the October 1984 issue of the *AT&T Bell Laboratories Technical Journal*. Devoted to UNIX and C, this issue includes the article by Dennis Ritchie on STREAMS, the concept behind AT&T's new UNIX networking products.

You can order this issue for \$27. Credit card holders may call 1-800-432-6600. If you are sending a check or money order, write

AT&T Bell Laboratories
Circulation Department
Room 1E335
101 J. F. Kennedy Parkway
Short Hills, NJ 07078

Q. Is there a method with C to poll a terminal keyboard in a XENIX environment? I have had no difficulties using `ioctl()` to set rawmode for unbuffered I/O, but I have been unsuccessful in discovering a method to duplicate the `KeyPressed` function in TurboPascal.

Thomas Fox
Clarksville, IN

A. Unfortunately, we haven't discovered the way either. Most of the gurus we talked with believe the `tty` driver must be modified or a new driver written.

The TurboPascal `KeyPressed` function returns a boolean `TRUE` value if a key has been pressed, otherwise `FALSE`. The function polls the keyboard "on-the-fly" and does not wait for a key to be pressed.

We would appreciate hearing from any of our readers who have an answer to this question. A portable UNIX answer would be even more welcome.

Q. I loved your article on common mistakes in your first issue (November 1984). However, having taught C to many beginners, I think you missed one other obvious mistake, = versus ==. This problem strikes most people who have programmed in BASIC.

Josh Ostern
San Francisco, CA

A. Absolutely. In BASIC, the equal sign is used for assignment and the logical test for equality. In C, one equal sign is the assignment operator, and two equal signs are the equality operator. This distinction confuses most BASIC programmers for a couple of weeks while they make their transition to C.

The worst time to forget the difference is when writing the expression for a `while` loop. Every C programmer at some time makes this mistake

```
while(i = 10)
```

and suddenly creates an infinite loop. Using `=` instead of `==` as the second expression in a `for` loop can have the same effect. These mistakes cause extreme head scratching until the problem is found.

Q. What is the origin of foo? I see it in my UNIX and C compiler documentation. I suspect there is a story behind it.

Bryan Caliburn
Bangor, ME

A. Foo comes from *foobar*, an acronym for "fouled-up beyond all recognition." Some localities prefer a different word than "fouled." Foobar is an English slang term, possibly introduced in the armed forces. The time of its tongue-in-cheek introduction to Bell Laboratories is unknown, but an ancient fourth edition of the UNIX documentation contained this word.

Q. Why doesn't my compiler set `argv[0]` to the program name? I read on page 111 of *The C Programming Language* by Brian Kernighan and Dennis Ritchie (K&R) that the first command line argument is the program that was invoked.

Samuel Wyatt
Palatine, IL

A. The fault is in the operating system, not the compiler. UNIX's shell preserves all command line arguments, allowing the C compiler to set the string pointed to by `argv[0]` to be the invoked program name.

With non-UNIX systems, the command interpreter may throw away the program name. This arrangement is true for CP/M, MS-DOS, AppleDOS, ProDOS, and many others. With these operating systems, the compiler has no way to divine the program name and is at the mercy of the host operating system to provide this service. No service, no program name.

Q. I'd like you to settle an argument. I believe that a `char` can be any size, from a six-bit byte to 30 bytes. My friend believes that a `char` is always the size of a native byte on a computer. Who is right?

Janet Harding
Washington, DC

A. I'll side with you.

K&R state that a byte is undefined in the language except for the `sizeof(char)`. A byte is defined as the space required to hold a `char`. This definition infers that a logical byte—not a machine byte—is needed to hold a character.

Few compilers use more than a native, physical byte to hold a character. It is, however, possible for a `char` to be any number of physical bytes. The only requirement placed on the compiler is that `sizeof(char)` be 1, and all other nonaggregate data types be in an integral multiple of `sizeof(char)`.

To settle another argument, nothing in K&R says an integer is two or four bytes large. For example, an `int` on a Cray I is three 8-bit bytes large (it is four bytes on a Cray II). Although Cray is not a household computer, this implementation does conform to the K&R rules.

Corrections and clarifications

- An error occurred in the article **epcat** by Glenn Ferrell in our February 1985 issue. When referencing Epson printers, the "TX" label should have been an *RX*. Epson manufactures an RX-70 and RX-80 printer, but does not have a TX-series. Also, the final line should have read that the program will work with the mentioned printer and *other* compilers with little or no change. The editor-in-chief apologizes for these two errors.

- Tom Fenwick, the author of the Aztec C compilers published by Manx Software Systems, took us to task over our January Quiz. The quiz hinged on how compilers with two-byte or four-byte integers would interpret the constant `0x8000`. All four-byte integer machines will interpret `0x8000` as a signed integer constant with the value of 32,768.

K&R states that if an octal or hexadecimal constant exceeds the largest unsigned integer, the constant is treated as a long integer. However, K&R does not explicitly state what is the data type of an octal or hexadecimal constant in larger-than-the-largest signed integer, but smaller-than-the-largest unsigned integer.

Tom's claim is that the number is a signed integer. The AT&T 8086 UNIX compiler treats `0x8000` as a signed integer. Tom further asserts that the K&R section where this information can be found is called *Integer Constants*, and the word `unsigned` only appears in the statement pertaining to exceeding the largest unsigned integer.

Tom's interpretation is backed by Larry Rosler, supervisor of the Language Systems Engineering Group at AT&T Bell Laboratories' UNIX Languages and Programming Development Environment. The Bell Lab's C compilers that have 16-bit integers treat `0x8000` as a signed integer.

The results of the assignment statements

```
x = 32768;  
x = 0x8000;
```

are different between compilers using two-byte and four-byte integers. However, the two statements should be identical for all compilers with two-byte integers; they are identical for all compilers with four-byte integers.

- Last month, we credited Dr. Thom Plum of Plum-Hall for the term *string literal*. However, we erred in reporting the X3J11 committee's proposals on the properties of string literals.

String literals are different from arrays of characters. The quoted material in the following line

```
printf("Where's the beef\n");
```

is considered a string literal. The quoted material in the following line which initializes `ca`

```
char ca[] = "Where's the beef\n";
```

is not a string literal, but is a character string. The difference falls between a double-quoted string used to initialize a character array and a double-quoted string used as an argument.

Also, string literals have a proposed type of array of `const char` and are not writable. Because the string literal is not writable, duplicate string literals may be pooled into one occurrence of the string.

Life and scope

(Continued from page 1)

A few compilers dynamically allocate space when each block is entered. The maximum automatic space needed for this function is 16K because the `goto` statement takes us around scope levels 2 and 3. We never reach the blocks that caused `buf2[]` and `buf3[]` to be physically allocated in our running program.

The impact of this compiler implementation trait is minimal. Because variables can only be declared at the start of a function or block, it is impossible to request more automatic variable storage when returning to a scope level. (We can't define `buf4[MAXARR]` upon returning to scope level 1.) Nor should code be written that always skips other sections of code. (Why did you write the skipped code in the first place?)

The only impact is when dynamically allocating more memory space using `malloc()` or `calloc()` in certain environments. If our fragment attempted to obtain some memory space upon return to scope level 1, the difference between compilers would be significant. Those compilers that create variables only when the function is entered could deny a valid request for what should be more free memory. Those compilers that create and kill variables as each block is entered would honor the request.

The best assumption about this implementation trait is that all automatic variables are created when the func-

tion is entered, not when each block is entered. This assumption will prevent surprises when moving to different compilers.

goto, life, and initialization

K&R state that it is bad practice to enter a block at any point other than the top. Figure 2 shows an example of this poor practice and begs a question. What is the return value of `whatLife()` if `where` is 1, 2, or 3?

For `where` equal to 1, the answer is 8. For `where` equal to 2 or 3, the answer is unknown.

Automatic variables are initialized when they are created *and* the block containing the variables is entered from the top. If `where` is 1, we technically enter that top of the block for scope level 2. We should pass through this level and level 3 without flaw. The answer should be 8 ($1 + 2 + (2 + 3)$).

If we jump to `label2`, we simultaneously jump into scope level 3. `y` may be alive, but not well. By jumping past the initialization of `y`, `y` could have any possible value. It would almost be an accident if `y` had a value of 2. However, `z` will be alive and have a value of 3. However, the result of $1 + 3 + \text{some value for } y$ is indeterminate.

The same logic applies for going to `label3`. `y` and `z` may be alive but uninitialized, and the result of the addition is unknown.

The best practice is to avoid this practice. If you must `goto` a block, either `goto` the top of the block and never

```
#define MAXARR 16384    /*      1/4 size of data segment      */

void showlife( )
{
    char buf1[MAXARR];           /* scope level 1 */
    goto label;

    {
        char buf2[MAXARR];       /* scope level 2 */
        {
            char buf3[MAXARR];   /* scope level 3 */
        }
    }

label:
    printf("I'm here\n");
}
```

Figure 1. Fragment testing when a compiler allocates automatic variables.

transcend more than one nested block, or just avoid `gotos` when automatic variables are declared within a jumped-to block. Better yet, use `goto` like curare—beneficial in small doses for certain life-threatening problems, yet fatal in any large dose.

Static and static

The difference between `static` (the storage class) and `static` (the keyword) is the connotation of life, initialization, and scope. From the standpoint of life, `static` is `static`. A `static` variable has an eternal lifetime (the life span of the program). A variable declared with a `static` keyword is of the static storage class and has this immortal lifetime.

From the standpoint of initialization, all static storage-class items are initialized once. If an initializer is given, it is used. If no initializer is given, the object is initialized to zeros. This rule includes `static` objects within a function. Other than permanence, this is the other deviance between `static` and `auto` variables. Automatic variables are initialized each time the block (function) is entered at the top.

Mentioned before, the keyword `static` is overworked. The keyword denotes both immortal storage and privacy for external objects. An external object defined with the keyword `static` is only available to other ob-

jects within the same file. This aspect of `static` externals allows the sheltering of external variables and functions from other files.

A `static` internal object (defined inside a function) is an immortal object (and by the rules of lexical scope, a `static` internal object is private to the function). A `static` external object is an immortal object that is private to the file. An external object without the keyword `static` is `static` (immortal, and, hopefully, no surprise).

`static` variables are used within functions for two reasons—to initialize an aggregate (array or structure), or to guarantee a one-time-only initialization of a private object. Last month's Soundex article (see page 9 of March's */c*) demonstrates the initialization of a `static` character array within a function.

Figure 3 shows a working function used in an unpublished program to test the Soundex routine. The function works on a link-list set of structures. The function starts with a static pointer (`nstruct` for *next structure*) to a structure of type `soundex` that is initialized to `NULL`. This allows us to test whether we will modify an existing list or start a new list. When entering the function a second or subsequent time, the pointer (`nstruct`) will have a non-`NULL` value and will signal that an established list should be manipulated. This is only possible because a `static` internal variable is initialized only once.

```
#define MAXARR 16384    /* 1/4 size of data segment */

int whatlife(whence)
int whence;
{
    int x = 1;           /* scope level 1 */

    switch(whence) {
        case 1:
            goto label1;
        case 2:
            goto label2;
        case 3:
            goto label3;
        default:
            break;
    }

label1: {                 /* scope level 2 */
    int y = 2,
    x += y;
label2: {                 /* scope level 3 */
    int z = 3,
label3:
    x += (y + z);
    }
    }
    return(x);
}
```

Figure 2. Fragment showing `goto` labels extending into nested blocks.

Figure 4 shows the skeletons of two sets of files. Figure 4a shows the definition of `x` in two different files. When the files are linked, the `x` in each file will remain unique to each file. `main()` in the first file will access only the `x` in the first file, and `f1()` will access the `x` in its file. There is no conflict between the two `x`'s.

Figure 4b shows the use of `static` to hide a function from any other files in the program. The function `f1()` in file 2 is defined as a `static` function. `f1()` in file 2 is available to `f2()` in the same file, but not to the first file. This example also shows how files with duplicate function names can be linked without drawing a re-definition error.

Although the example in Figure 4b is trivial, a more positive use of `static` functions is to hide functions providing a special service to other functions in the same file. When writing a program in MS-DOS that performs searches for filenames down a directory path, it was necessary to create functions that set and restore DOS' data transfer area, the area where the filename is returned. `getdta()` and `setdta()` functions performed these tasks. Because these functions had peculiarities specific to that file (i.e., it wasn't desirable to make these functions independent library functions), the two functions were defined as `static`. This step prevents name clashing if another program defining `getdta()` or `setdta()` and using this module prevents these functions' independent use.

External and extern

External refers to an object defined (or declared) outside of a function. By rule, function definitions are external (you can't define a function within a function). However, `extern`, another overworked keyword in the C language, affects scope, storage class, and can determine if an identifier is used for a definition or declaration.

There is an `extern` storage class (the opposite of `auto`) for nonfunction objects; an `extern` storage class (the opposite of `static`) for function definitions; and an internal `extern` declaration with the same syntactic meaning as an external `extern` declaration, but the internal `extern` declaration may have different connotations.

This entire area is one of the ambiguities in K&R that seemingly contradicts itself. The contradiction comes from K&R stating that all external objects have the `extern` storage class unless the keyword `static` is used. However, an external, nonfunction object in a multifile program must be externally declared once and only once in one of the files without the keyword `extern`. A further contradiction comes when K&R state that an external definition using the word `extern` indicates that the storage for the object is in a different file—which instead makes it a declaration. This logic implies that a nonfunction object cannot be defined if the definition contains the keyword `extern`.

```
typedef struct sounder {          /* structure for list */
    char *name,                  /* name to soundex */
    int sxval;                   /* soundex value */
    struct sounder *next;        /* pointer to next entry in list */
} SXNODE, *SXPTR;

SXPTR ninstall(s)                /* make and install a node in the list */
char *s,                         /* return pointer to struct or NULL */
{
    static SXPTR nstruct = NULL; /* current structure in list */
    SXPTR sp,                   /* temp structure pointer */
    SXPTR salloc();              /* struct space allocator */

    if((sp = salloc(1)) != NULL) { /* got room */
        sp->name = s;             /* point to the name */
        sp->sxval = soundex(s);    /* put in the soundex value */
        if(nstruct == NULL)       /* first time in? */
            nstruct = sp;         /* nstruct points to saved struct */
        else {
            nstruct->next = sp;    /* point to next struct */
            nstruct = sp;         /* same here */
        }
    }
    return(sp);                  /* return the pointer or NULL */
}
```

Figure 3

The best way to view `extern` is with the keyword used only in declarations. When `extern` is used for nonfunction, internal declarations, `extern` states that the nonfunction object is a global object (external) defined among one of the files—maybe the same file. For external nonfunction objects, the keyword `extern` states that the object is defined elsewhere, but implies the object is defined in another file—although the object could be defined in the same file.

When used in function declarations, inside or outside of a function, `extern` also implies that function is defined in a different file.

One other common use of the `extern` keyword is in a function definition. This use explicitly states that the function is nonstatic. This distinction makes the function public for use by other files.

Some examples of `extern`'s use are in Figure 5. The `extern int x;` in Figure 5a refers to the `x` declared in the program between `main()` and `f1()`. This declara-

tion shows the use of `extern` to refer to an object defined later in the file.

Figure 5b shows the more conventional uses of `extern`. The `extern int x` within refers to the global `x` in file 2. The `extern int f2()` is a global declaration that states `f2` is defined somewhere. In this declaration, the `extern` implies the function is defined in a different file. The use of the keyword `extern` is unnecessary because all function declarations, regardless of location, are `extern` by default. In this case, the keyword `extern` becomes a documentation aid implying to the programmer that the function `f2()` is defined in a different file.

One final note about `extern`: Except when used in a function definition, an `extern` object cannot be initialized. Because `extern`'s use for nonfunction objects denotes that the storage for an item is performed elsewhere, you cannot give this object an initializer. You can only initialize an object when the object is defined.

Demonstrations of static externals

a. two x example

File 1

```
static int x = 3;

main()
{
    extern int f1();
    x += f1();
    printf("x = %d\n", x);
}
```

File 2

```
static int x = 5;

int f1()
{
    x *= 2;
    return x;
}
```

b. use of static functions

File 1

```
static int x = 2;

main()
{
    extern int f1();
    x += f1();
    f2(x);
    printf("x is %d\n", x);
    exit(0);
}

int f1()
{
    extern int f2();
    int y = 4;
    return(x * y);
}
```

File 2

```
static int x = 5;

static int f1()
{
    return (x * *);
}

int f2(x)
int x;
{
    x += f1();
    printf("x is %d\n", x);
}
```

Figure 4

Except in a function definition, `extern` denotes a declaration—not a definition.

Summary

Life, the run-time concept, affects when objects are brought into being. We discussed the two storage classes, static and automatic. As shown previously, automatic variables may be brought to life sooner than we might believe.

Scope, the compile-time concept, affects when an activity can take place with an object. We showed how

`extern` is used in declarations to declare objects before their definitions, or to declare objects in different files.

We briefly reviewed initialization of variables and discussed the to-be-avoided practice of `goto`ing a nested block.

Life and scope are two C aspects learned by the programmer—not through formal study, but by experience. This article has attempted to examine the issues most programmers handle on an intuitive basis.

Use of the keyword `extern`

5a. Single file use of `extern`

```
extern int x;

main()
{
    x += 5;
    printf("x is now %d\n", x);
    x += f1(),
    printf("x is now %d\n", x);
    exit(0);
}

int x;

int f1()
{
    return(x * 4);
}
```

5b. Use of `extern` for crossing files

File 1

```
extern int x;
extern int f2(),

main()
{
    x += 5;
    x += f1(),
    x += f2(),
    printf("x is now %d\n", x);
    exit(0);
}

int f1()
{
    return(x * 4);
}
```

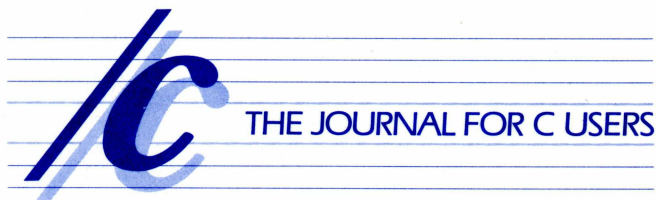
File 2

```
int x;

int f2()
{
    int y = 5;
    return(y * y + x);
}
```

Figure 5

Subscribe to . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, /c is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (1-800-227-7999) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(FOLD HERE)

YES, I want /c.

- ☐ Send me 12 monthly issues for \$60 (outside U.S.A., \$80)*
- ☐ Send me 24 monthly issues for \$105 (outside U.S.A., \$135)*

*All money must be payable in U.S. funds.

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

Signature _____

(for credit card)

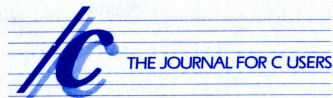


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

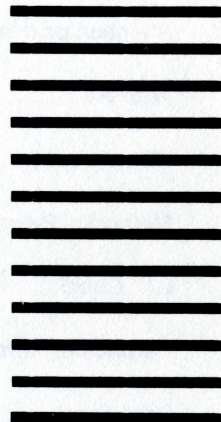
First Class Permit No. 9918 Indianapolis, IN

Postage will be paid by addressee



THE JOURNAL FOR C USERS

P.O. Box 50507
Indianapolis, IN 46250

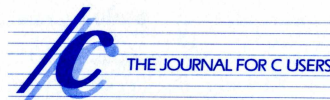




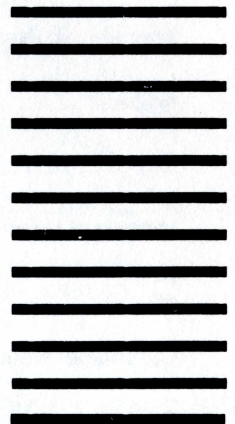
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Call Permit No. 9918 Indianapolis, IN

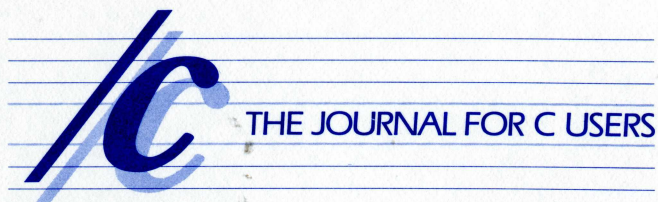
Postage will be paid by addressee



P.O. Box 50507
Indianapolis, IN 46250



Subscribe to . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, /c is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (1-800-227-7999) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(FOLD HERE)

YES, I want /c.

- ☐ Send me 12 monthly issues for \$60 (outside U.S.A., \$80)*
- ☐ Send me 24 monthly issues for \$105 (outside U.S.A., \$135)*

*All money must be payable in U.S. funds.

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

Signature _____

(for credit card)

ANSI committee releases tentative form of C language draft

In its March meeting in Chapel Hill, North Carolina, the ANSI X3J11 Committee, the C Standards Committee, voted to informally release the current draft of its work through ANSI. The draft will be available sometime in April.

Still subject to fine-tuning and other changes, the current draft will be released as an information bulletin. An ANSI *information bulletin* is simply a mechanism for parlaying information to interested parties. In the case of the C language draft, it allows interested parties to obtain a copy and to make informal comment to the committee.

Actively seeking outside comment and suggestion, the committee voted to use the information bulletin mechanism rather than make a formal release of the draft. By soliciting informal comment, the committee can then receive inquiries and suggestions without the obligation of a written response. This method differs from the release of a *draft-proposed American National Standard* (dpANS) requiring written response to all suggestions and comments. Because the draft still contains some unresolved "rough spots," the committee felt that a six-month period of informal comment was a significant step toward finalization.

The current timetable, which remains fluid, is to resolve further issues during the June and September quarterly meetings, and to attempt to vote the draft to a proposed ANSI standard in the December meeting. If this timetable is met, the proposed draft would go

through a one-year comment period and then could be adopted as an ANS (American National Standard) in December 1986.

Individuals interested in obtaining the information bulletin should contact the X3 Secretariat at:

Computer and Business Equipment
Manufacturers Association
311 First Street NW
Suite 500
Washington, DC 20001

No price for the draft has been set but it is expected to be in the range of \$25. Because the bulletin must go through a 30-day X3-balloting period, it should be available near the end of April.

Our customary caution is given. The draft is a draft and is not a standard until formally adopted. Do not expect implementors to adhere to the draft. Nor should you expect any compiler to advertise adherence to the draft (which is barred by ANSI rules). All times are subject to change, and portions in the draft may change. The members of the committee appreciate comments, but cannot respond individually.

Next month, /c will examine the status of the C language draft and review the differences between K&R, C, and the proposal. We'll also highlight those proposal portions varying from K&R C components that are already common practice with the C language.

Quickie

Does C have formal subroutines?

The answer is on page 16.

fd versus fp

Many C newcomers are confused by the differences between the two sets of file functions. This article explains some of the differences.

The starting point

The major difference between the two sets of file input and output functions comes from C's origin in the UNIX operating system. The UNIX operating system provides system routines for the routines using file descriptors (fds). The C standard library under UNIX provides the additional routines for file pointers (fps).

To discuss the differences, we'll use two terms—operating system I/O, or *osio*, for the functions and var-

iables associated with fds, and standard I/O, or *stdio*, which applies to functions and variables associated with fps.

OS services

Osio services in UNIX use something called a file descriptor. The file descriptor, usually abbreviated to *fd*, is typically a signed integer, although a valid file descriptor is never negative. These services are obtained by the program issuing a request to the underlying operating system. This technique is commonly called a *system call*.

`open()` and `creat()` form the cornerstone for the set of

operating system services. Under UNIX these functions are defined as

```
int open(name, flags[, mode])
char *name;
int flags;
[ int mode; ]

int creat(name, mode)
char *name;
int mode;
```

name is a pointer to a filename. C compilers under operating systems that use hierarchical directories allow **name** to include the path to the file. The integer variable **flags** governs whether the file is open for input (reading), output (writing), or input and output. Other options may include where the file should be created (the same as using `creat()`), or truncated (the file cleared out before reuse), or if information sent to the file should be appended to the end of the file. Most compilers offer a set of symbolic names for **flags** that are ORed together. The names can be found in a file called `fcntl.h`.

mode is optional for `open()`, mandatory for `creat()`. Under UNIX, **mode** affects who may use the file. For non-UNIX compilers, **mode** may have a completely different use.

`open()` sets up a file for input and output. Under UNIX, files and devices "look alike," and `open()` is also used for devices. Previously, `open()` would not work if the file **name** did not exist. The `O_CREAT` flag for `open()` allows new files to be created without a specific call to `creat()`.

The return value for these two system calls is an integer file descriptor. Until the file or device is closed, this integer uniquely identifies the file or device. The returned integer is normally assigned to a variable such as `fd`, `fd1`, `fd2`, `fdin`, `fdout`, or a similar name. (C programmers do not get very imaginative when it comes to the names for file descriptors.)

The OS set of functions that works with the file descriptor returned by `open()` or `creat()` is:

```
int close(fd)
int dup(fd)
int fcntl(fd, command, arg)
int ioctl(fd, command, arg)
lockf(fd, function, nbytes)
long lseek(fd, offset, position)
int read(fd, buf, nbytes)
int write(fd, buf, nbytes)
```

The first function parameter is always a file descriptor. These functions do not use a filename. The functions only use the integer file descriptor returned from `open()` or `creat()`.

The four most commonly used functions on this list are `read()`, `write()`, `lseek()`, and `close()`. `read()` gets information from the file; `write()` sends information to the file; `lseek()` allows random movement within

the normally sequentially accessed file; and `close()` closes out activity on the file.

To understand why these OS functions are sometimes called *nonbuffered* (a misnomer) functions on non-UNIX systems, consider the second parameter for `read()` and `write()`. **buf**, a character pointer, is a descriptive name for the area in memory to hold the incoming or outgoing information. The programmer is responsible for setting up and providing the buffer for `read()` and `write()`. Normally, the buffer space comes from a defined character array, such as `char buf[BUFSIZ];`, or a call to `malloc()` (the run-time memory allocation function) within the executing program.

Stdio services

Standard I/O services are not operating system calls but functions from the C standard library. The stdio family of functions parallel the OS I/O family.

The cornerstone of stdio is `fopen()`. `fopen()` is defined as

```
FILE *fopen(name, fmode)
char *name;
char *fmode;
```

As with `open()`, the first parameter for this function is a filename. The second parameter, **fmode**, is a pointer to a string that tells the compiler whether this file will be read, written to, read and written, or if written information should be appended to the end of the file. Many non-UNIX compilers also allow **fmode** to specify if some end-of-line transformation (such as changing a newline to carriage return-newline and back) should take place.

`fopen()` returns a pointer to `FILE`. `FILE` is normally a typedefed structure declared in `stdio.h`. The structure contains several pieces of information about the file.

The pointer to `FILE` is normally called an **fp** (file pointer). A valid **fp**, like any valid pointer, is guaranteed not to be 0. Conforming to practice, the file pointers are assigned to such variables as `fp0`, `fp1`, `fpin`, and `fpout`.

There are more than 27 functions and/or macro functions in UNIX that work with `fopen()`. Some stdio functions are `gets()`, `scanf()`, `printf()`, `fread()`, `fwrite()`, `fseek()`, `fclose()`, and `setbuf()`. Each of these stdio functions uses an **fp** rather than an **fd**. `getchar()` and `putchar()` are normally macro definitions that actually call either `getc()` and `putc()`, or `fgetc()`, or `fputc()`. You should check the file `stdio.h` for these definitions.

The relationship

There is a definitive relationship between operating system I/O and standard I/O. Standard I/O is performed by calling operating system I/O. In essence, there is only one true set of routines to move information between the program and the operating system. However, stdio must add a layer of routines to buffer input and output for the file.

A simple example is `getc()`, a function that gets a single character from a file. When you call `getc()`, does the program ask the operating system for a single character? Probably not.

When the file is `fopen()`ed and the first character is read from the file, a buffer large enough to hold a "block" of characters from the disk is created somewhere in memory. The size of the block varies between computers and devices. Usually, the "best" size for the block (such as the size of a disk's sector) is used.

The library routines then invoke `read()` to bring in a block from the disk. Each time `getc()` is called, a character is transferred from the hidden buffer back to the invoking function. When the block is exhausted, a new block is read into the buffer using `read()`.

All stdio functions, except `fread()` and `fwrite()`, use the hidden buffer. This method is how the standard library transforms various-size blocks of information on the disk into ordered sets of information for functions like `fgets()` or `fscanf()`.

An additional headache for the compiler writer is `ungetc()`, the function that allows a previously gotten character to be pushed back to the file. The stdio routines must track whether a character has been pushed back. If so, the pushed back character must be the next character from the file.

Some standard files

UNIX provides three default fps. `stdin` is the file pointer used for standard input, usually the keyboard. `stdout` is the standard output device name, usually the terminal or video screen. `stderr` is the standard device for displaying errors. `stderr` is normally the same device as `stdout`. Logically, `stderr` is different from `stdout`. When the output of a program is redirected, only `stdout` is redirected, not `stderr`. This redirection allows the display of error messages to the normal display device.

Three file descriptors are normally also reserved by UNIX. These fds are 0 for standard input, 1 for standard output, and 2 for standard error.

Some operating systems and compilers extend these standards. Some compilers support a fourth device, fd 3 or `stdaux`, for the auxiliary device (normally a communications port), and fd 4 or `stdlst` for standard list (the printer).

The link

There is a function that links stdio and osio. The function is `fileno()`, which returns the integer fd used by the stdio routines for a particular fp. The function call is

```
int fileno(fp)
FILE *fp;
```

The fp is a file pointer used as the function argument to `fileno()`.

This function can be used to find which fd is associated with the internal use of an fp. We will point out an improper use of `fileno()` subsequently.

Mixing osio and stdio

There are two cardinal rules about osio and stdio.

1. *Don't use file pointers with osio routines and don't use file descriptors with stdio routines.*

Rule 1 is simple. Osio routines use a file descriptor—an integer number. Stdio routines use a file pointer, usually a pointer to a structure of type `FILE`. (You can find out if this rule is true for your compiler by examining the contents of `stdio.h`, the standard I/O header file. Look for the structure or array named in the typedef for `FILE`.)

Sending an integer to a function that wants a pointer to a structure can give interesting results (and send your program on the proverbial "trip west"). Sending a structure pointer to a function expecting an integer can give similarly interesting results.

The problem worsens for programs that don't perform error checking on `read()` or `write()` (which under ideal circumstances will return a -1 on the error of passing an fp), or on the stdio functions such as `fgets()` or `fputs()` (which might return `NULL` if the circumstances are right when the fd is used). Even with checks on the return values of these functions, programs can still "go west," as the I/O functions attempt to read or write information from "who-knows-what" files.

2. *Avoid mixing the osio function with stdio function for the opened (or fopened) file.*

Rule 2 reflects a property of stdio. Stdio normally uses a hidden buffer. For example, a file is `fopen()`ed. Several `fgets()` have been performed on an `fopen()`ed file. The program gets the fd of the file using `fileno()` and performs a `read()`. You should be astonished if this process works.

Reading or writing to a file via the `read()` or `write()` osio functions may bypass the hidden buffer. Stdio may have some characters from the file in its hidden buffer. Performing a `read()` bypasses stdio (and the characters remaining in the hidden buffer) and grabs more characters from the file. `read()` may have destroyed the synchronization of the file. `read()` has literally gotten characters "out-of-order" as the next set of characters to be processed is still in the hidden buffer. Those programmers who demand "file integrity" should shudder at this practice.

The programmer can mix osio and stdio functions on a character-oriented device, provided stdio *does not buffer* the input or output. However, this practice is operating system-, compiler-, and device-specific and should be avoided.

The programmer can also `open()` a file, use osio routines, `close()` the file, `fopen()` the file and use stdio

routines. The opposite is true. The file could be fopen(), manipulated using stdio functions, fclose()ed, open()ed, and osio routines used. This procedure does not violate rule 2. When a file is closed, it may be opened again using a different set of I/O functions and those I/O functions in the family (osio or stdio) may be freely used.

The tables

Table 1 is a list of osio functions. Each of these use an fd. Table 2 is a list of stdio functions and macros, each of which is related to an fp. Unlike Chinese menus, from the point of opening a file until the file is closed, choose only from column A or from column B, but not from both.

Table 1

Osio functions

(Use or return a file descriptor, an fd)

open/close

```
int open(name, flags[, mode])
int creat(name, mode)
int close(fd)
```

open file
creates and opens a file
closes file

file access

```
int read(fd, buf, nbytes)
int write(fd, buf, nbytes)
long lseek(fd, offset, position)
```

reads from file
writes to file
random movement within file

miscellaneous

```
int dup(fd)
int fileno(fp)
int fcntl(fd, command, arg)
int ioctl(fd, command, arg)
lockf(fd, command, nbytes)
```

creates another fd for the file
returns fd of an fp
governs characteristics of file
governs characteristics of file
file locking (System V.2.2)

key

```
arg
buf
command
fd
flags
fp
mode
name
nbytes
offset
position
```

argument to function
pointer to char. array
action to be performed
file descriptor
open/create flags
pointer to FILE
permission modes (os specific)
char. pointer to a filename
number of bytes
number of bytes to move
orientation for the seek

Table 2

Stdio functions

(Functions and macros that use a pointer to FILE an fp)

open/close

```
FILE *fopen(name, fmode)
FILE *freopen(name, fmode, fp)
FILE *fdopen(fd, fmode)
fopen()s an open()ed file
FILE *tmpfile()
int fclose(fp)
```

opens a stream
reopens a stream (fp)

opens a temporary file
closes a stream

character/word input

```
int getchar()
intgetc(fp)
int fgetc(fp)
int getw(fp)
int ungetc(c, fp)
```

get a character - stdin (M)
get a character (M)
get a character
get a "word"
push back a character

(Continued on next page)

character/word output

```
int putchar(c)
int putc(c, fp)
int fputc(c, fp)
int putw(w, fp)
```

put a character - stdout (M)
put a character (M)
put a character
put a "word"

line input/output

```
char *gets(buf)
char *fgets(buf, nbytes, fp)
int puts(buf)
int fputs(buf, fp)
```

get a line - stdin
get a line
put a line - stdout
put a line

formatted input

```
int scanf(format, args ...)
int fscanf(fp, format, args ...)
int sscanf(buf, format, args ...)
```

formatted input - stdin
formatted input
formatted input - in-memory string

formatted output

```
int printf(format, args ...)
int fprintf(fp, format, args ...)
int sprintf(buf, format, args ...)
```

formatted output - stdout
formatted output
formatted output - in-memory string

formatted output - variable list

```
int vprintf(format, va_list)
int vfprintf(fp, format, va_list)
int vsprintf(buf, format, va_list)
```

formatted output - stdout
formatted output
formatted output - in-memory string

miscellaneous

```
char *ctermid(buf)
int ferror(fp)
int feof(fp)
void clearerr(fp)
int fileno(fp)
int fread(fp, buf, nbytes)
int fwrite(fp, buf, nbytes)
void setbuf(fp, buf)
int setvbuf(fp, buf, iotype, nbytes)
char *tmpnam(buf)
char *tempnam(dir, pfx)
```

gets terminal name
checks error indicator for stream
checks if EOF has been encountered
clears error indicator
returns fd of an fp
stdio equivalent of read()
stdio equivalent of write()
sets location of buffer for stream
sets location and characteristics of buffer for stream
creates a safe name for a temporary file
creates a safe name starting with pfx for a temporary file

key

```
arg
buf
c
command
dir
fd
flags
fmode
format
fp
iotype
name
nbytes
offset
pfx
position
stream
a_list
w
```

argument to function
pointer to char. array
integer used for character input/output
action to be performed
directory name
file descriptor
open/create flags
file mode (os specific)
pointer to array holding formatting string
pointer to FILE
buffered/nonbuffered I/O type
char. pointer to a file name
number of bytes
number of bytes to move
prefix to file name (char *)
orientation for the seek
an input/output file
variable argument list
integer used for "word" input/output

The following is the output of the program we wrote to sum the ASCII characters in the C keywords. The list reflects many common local keywords.

of keywords is 42

asm	321	for	327
define	619	goto	441
endif	518	if	207
entry	562	int	331
fortran	764	long	432
include	740	register	869
ifdef	510	return	672
ifndef	620	short	560
line	424	signed	634
pragma	632	sizeof	656
undef	530	static	648
auto	441	struct	677
break	517	switch	658
case	412	typedef	753
char	414	union	553
const	551	unsigned	861
continue	869	void	434
default	741	volatile	864
do	211	while	25300
double	635		
else	425		
extern	662		
float	534		

matches

auto	goto	441
continue	register	869

sumkey.c

```

/*      program to produce ASCII values of keywords and should matches */
#include <stdio.h>
#define EOS '\0'
char *keyword[] = {
    /* local, new, and old keywords and preprocess directives */
    "asm", "define", "endif", "entry", "fortran", "include",
    "ifdef", "ifndef", "line", "pragma", "undef",

    /* c keywords */
    "auto", "break", "case", "char", "const", "continue", "default",
    "do", "double", "else", "extern", "float", "for", "goto", "if",
    "int", "long", "register", "return", "short", "signed", "sizeof",
    "static", "struct", "switch", "typedef", "union", "unsigned",
    "void", "volatile", "while",
    0
};

main()
{
    int i, j, keywords, hit;
    char *cp, *mp;
    int *ip;
    char *calloc();

    /* find number of keywords + 1 */
    for(keywords = 0; keyword[keywords]; ++keywords);
    printf("# of keywords is %d\n", keywords);

```

(Continued on next page)

```
/* get the dynamic storage for the int array to hold sums */
/* and char array to hold matches */
ip = (int *)calloc(keywords - 1, sizeof(int));
mp = calloc(keywords - 1, sizeof(char));
if(ip == NULL) {
    /* wrongo! */
    printf("not enough room to calloc() space.\n");
    exit(1);
}
for(i = 0; i < keywords; ++i) {
    cp = keyword[i];
    while(*cp)
        ip[i] += *cp++;
}
for(i = 0; i < keywords; ++i)
    printf("%10s %5d\n", keyword[i], ip[i]);
printf("\nmatches\n");
for(i = 0; i < keywords - 1; ++i) {
    if(mp[i])
        continue;
    /* for keyword + 1 to end */
    for(j = i + 1; j < keywords; ++j) {
        if(ip[i] == ip[j]) {
            if(!mp[i]) {
                printf("%12s ", keyword[i]);
                mp[i]++;
            }
            printf("%12s ", keyword[j]);
            mp[j]++;
        }
    }
    if(mp[i])
        printf(" %d\n", ip[i]);
}
exit(0);
}
```

Functions and articles wanted

/c is looking for authors of functions and articles on the following subjects:

- Tips on the C language
- C Programming techniques
- UNIX
- Application and programming tools
- Discussion and explanations of source code

The tone and content should be "moderately" technical. If any code is specific to an environment or compiler, the host environment or compiler should be clearly identified.

Writers should include a brief outline; clips of current works (if any); a current resume (if available); a telephone number; and a self-addressed, stamped envelope. We reply to submissions as soon as possible. Submissions used are paid on publication of the article or item.

Submit double-spaced typewritten copy with standard margins, or submit single-spaced text on CP/M (8-inch) or IBM PC-compatible (5 1/4-inch) diskettes.

Accompanying photos, sketches, diagrams, graphs, and charts are encouraged.

Length of articles: 300-1,000 words for news, 1,000-5,000 words for features, but no set length for source code.

Submissions and inquiries should be sent to

/c Editor
c/o Que Publishing, Inc.
7999 Knue Road
Indianapolis, IN 46250

All material accepted for publication in /c becomes property of Que Corporation.

C quiz

This and every month /c offers a quiz. The first correct response received at our /c offices receives a small prize. In the case of a tie, the earliest postmark wins. All entries must be mailed.

February quiz

C currently has 29 keywords if you include the new data type void. The ANSI X3J11 committee's current draft deletes the keyword entry and adds const, volatile, and signed.

Summing the ASCII value for each character in a keyword, which keywords have the same sum?

The answer:

Of the 31 keywords, only two sets of words have identical ASCII sums. The keywords are `auto` and `goto` (sum of 441), and `continue` and `register` (sum of 869).

Hopefully, you wrote a program to compute the sums. The program we wrote appears in this issue of /c.

Interestingly, we also put back the keyword `entry`, added all old and new preprocessor commands (such as `include` and `pragma`) and "local" keywords (such as `fortran` and `asm`), and reran the program. No new matches were found.

We congratulate two winners of the February quiz, Ralph Keeton of Dallas, Texas, and Steve Sherrill of Atlanta, Georgia. Both winners will receive a copy of *Common C Functions* by Kim Brand, and our bonus of \$25 for no correct answers to February's Quiz.

April quiz:

Given that i is an integral variable, how many possible, fully expanded definitions and declarations exist for i? For example,

```
extern int i;
register unsigned int i;
short int i;
```

are three possible declarations or definitions. Do not include definitions/declarations that use the proposed keyword `signed`, or use definitions/declarations based on accepted "shorthand" notations, such as `long i`. Assume any definition/declaration is eligible for CPU register use.

Send your answers to:

/c Quiz
c/o Que Publishing, Inc.
7999 Knue Road
Indianapolis, IN 46250

The answer to April's quiz will appear in June's issue.

D language

This month's exciting new extensions cover the D preprocessor.

`#exclude filename`

Allows the programmer to specify which files on the disk are *not* to be included in the compilation.

`#dctorp`

A preprocessor directive to ignore all error flags and continue compilation (short for "damn the torpedoes").

Our thanks to Harry W. Kroeger, Jr., who has attained immortality and \$10 for his contributions to the illogical successor of C.

To join the ranks of the immortal ones (and get \$5 per item in the process), send your suggestions to

/c D
c/o Que Corporation
7999 Knue Road
Indianapolis, IN 46250

Regrettably, all suggestions printed become property of Que Corporation. Also, your name will be printed. (What price, immortality?)

Quickie answer

The answer is no.

By definition, a formal subroutine is called using a `go` subroutine instruction, such as FORTRAN and BASIC's `GOSUB` or assembler `JSR` instruction. A subroutine *never* returns a value.

A function is also called, but *always* returns a value—even though the value may be "junk."

BASIC and assembler use only subroutines. FORTRAN allows both subroutines and function calls. C does not allow formal subroutines—only function calls. C has no `GOSUB` instruction. All C functions, including void functions, return something. The value returned may be meaningless, but some value is returned.

Letters

/c enjoys hearing from its readers. If you have a question or comment, send a note to the editor. You'll find the address on page 2.